Kotlin - Class and Objects

Kotlin supports both functional and object-oriented programming. While talking about functional features of Kotlin then we have concepts like functions, higher-order functions and lambdas etc. which represent Kotlin as a functional language.

Kotlin also supports Object Oriented Programming (OOP) and provides features such as abstraction, encapsulation, inheritance, etc. This tutorial will teach you all the Kotlin OOP features in simple steps.

> Object oriented programming (OOP) allows us to solve the complex problems by using the objects.

Kotlin Classes

A class is a blueprint for the objects which defines a template to be used to create the required objects.

Classes are the main building blocks of any Object Oriented Programming language. A Kotlin class is defined using the **class** keyword. Following is the syntax to create a Kotlin Class:

> A Kotlin class declaration is similar to Java Programmig which consists of a **class header** and a **class body** surrounded by curly braces.

```
class ClassName { // Class Header

   //
   // Variables or data members
   // Member functions or Methods
   //
   ...
   ...
}
```

By default, Kotlin classes are **public** and we can control the visibility of the class members using different modifiers that we will learn in **Visibility Modifiers** .

Kotlin Objects

The objects are created from the Kotlin class and they share the common properties and behaviours defined by a class in form of data members (properties) and member functions (behaviours) respectively.

The syntax to declare an object of a class is:

```
var varName = ClassName()
```

We can access the properties and methods of a class using the **.** (dot) operator as shown below:

```
var varName = ClassName()

varName.property = <Value>
varName.functionName()
```

Example

Following is an example where we will create one Kotlin class and its object through which we will access different data members of that class.

```
class myClass {
   // Property (data member)
   private var name: String = "Tutorialspoint.com"

   // Member function
   fun printMe() {
      print("The best Learning website - " + name)
   }
}
fun main(args: Array<String>) {
   val obj = myClass() // Create object obj of myClass class
   obj.printMe() // Call a member function using object
}
```

The above piece of code will yield the following output in the browser, where we are calling **printMe()** method of **myClass** with the help of its own object **obj**.

```
The best Learning website - Tutorialspoint.com
```

Kotlin Nested Class

By definition, when a class has been created inside another class, then it is called as a **nested class**.

Kotlin nested class is by default static, hence, it can be accessed without creating any object of that class but with the help of **.** dot operator. Same time we cannot access members of the outer class inside a nested class.

Following is the simple syntax to create a nested class:

```
class OuterClass{
   // Members of Outer Class
```

```
    class NestedClass{
       // Members of Nested Class
    }
 }
```

Now we can create an object of nested class as below:

```
 val object = OuterClass.NestedClass()
```

Example

Following is the example to show how Kotlin interprets a nested class.

```
 fun main(args: Array<String>) {
    val obj = Outer.Nested()

    print(obj.foo())
 }
 class Outer {
    class Nested {
       fun foo() = "Welcome to The TutorialsPoint.com"
    }
 }
```

When you run the above Kotlin program, it will generate the following output:

```
 Welcome to The TutorialsPoint.com
```

Kotlin Inner Class

When a nested class is marked with a keyword **inner**, then it will be called as an **Inner class**. An inner class can be accessed by the data member of the outer class.

Unlike a nested class, inner class can access members of the outer class. We cannot directly create an object of the inner class but it can be created using the outer class object.

Following is the simple syntax to create an inner class:

```
 class OuterClass{
    // Members of Outer Class
    class inner InnerClass{
       // Members of Inner Class
    }
 }
```

Now we can create an object of inner class as below:

```
val outerObj = OuterClass()
val innerObj = outerObj.InnerClass()
```

Example

Following is the example to show how Kotlin interprets an inner class.

```
fun main(args: Array<String>) {
   val obj = Outer().Inner()

   print(obj.foo())
}
class Outer {
   private val welcomeMessage: String = "Welcome to the TutorialsPoint.com"
   inner class Inner {
      fun foo() = welcomeMessage
   }
}
```

When you run the above Kotlin program, it will generate the following output:

```
Welcome to The TutorialsPoint.com
```

Anonymous Inner Class

Anonymous inner class is a pretty good concept that makes the life of a programmer very easy. Whenever we are implementing an interface, the concept of anonymous inner block comes into picture. The concept of creating an object of interface using runtime object reference is known as anonymous class.

Example

Following is the example to show how we will create an interface and how we will create an object of that interface using Anonymous Inner class mechanism.

```
fun main(args: Array<String>) {
   var programmer :Human = object:Human { // Anonymous class
      override fun think() { // overriding the think method
         print("I am an example of Anonymous Inner Class ")
      }
   }
   programmer.think()
}
interface Human {
   fun think()
}
```

When you run the above Kotlin program, it will generate the following output:

```
I am an example of Anonymous Inner Class
```

Kotlin Type Aliases

Kotlin Type Aliases means a way to give an alternative name to an existing type. Type alias provides a cleaner way to write a more readable code.

Consider a following function which returns a user info first name, last name and age:

```
fun userInfo():Triple<String, String, Int>{
   return Triple("Zara","Ali",21)
}
```

Now we can a type alias for the given **Triple** as follows:

```
typealias User = Triple<String, String, Int>
```

Finally the above function can be written as below, which looks more clean than above code:

```
fun userInfo():User{
   return Triple("Zara","Ali",21)
}
```

Example

Following is the complete working example to show the usage of type alias in Kotlin:

```
typealias User = Triple<String, String, Int>

fun main() {
    val obj = userInfo()

    print(obj)
}

fun userInfo():User{
   return Triple("Zara","Ali",21)
}
```

When you run the above Kotlin program, it will generate the following output:

```
(Zara, Ali, 21)
```

Quiz Time (Interview & Exams Preparation)

Q 1 - By default Kotlin classes are public:

A - Yes

B - No

Q 2 - Kotlin allows to access members of the outer class inside a nested class.

A - True

B - False

Q 3 - Kotlin allows to access members of the outer class inside a inner class.

A - True

B - False

Q 4 - What is Kotlin Type Aliases?

A - A way to name an existing type

B - A way to create a new data type

C - A way to define a variable

C - None of the above